Computational semantics in the Natural Language Toolkit

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
In Proceedings of the Australasian Language Technology Workshop

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Computational Semantics in the Natural Language Toolkit

Ewan Klein
School of Informatics
University of Edinburgh
Scotland, UK
ewan@inf.ed.ac.uk

Abstract

NLTK, the Natural Language Toolkit, is an open source project whose goals include providing students with software and language resources that will help them to learn basic NLP. Until now, the program modules in NLTK have covered such topics as tagging, chunking, and parsing, but have not incorporated any aspect of semantic interpretation. This paper describes recent work on building a new semantics package for NLTK. This currently allows semantic representations to be built compositionally as a part of sentence parsing, and for the representations to be evaluated by a model checker. We present the main components of this work, and consider comparisons between the Python implementation and the Prolog approach developed by Blackburn and Bos (2005).

1 Introduction

NLTK, the Natural Language Toolkit, is an open source project whose goals include providing students with software and language resources that will help them to learn basic NLP. NLTK is implemented in Python, and provides a set of modules (grouped into packages) which can be imported into the user’s Python programs.

Up till now, the modules in NLTK have covered such topics as tagging, chunking, and parsing, but have not incorporated any aspect of semantic interpretation. Over the last year, I have been working on remedying this lack, and in this paper I will describe progress to date. In combination with the NLTK parse package, NLTK’s semantics package currently allow semantic representations to be built compositionally within a feature-based chart parser, and allows the representations to be evaluated by a model checker.

One source of inspiration for this work came from Blackburn and Bos’s (2005) landmark book Representation and Inference for Natural Language (henceforth referred to as B&B). The two primary goals set forth by B&B are (i) automating the association of semantic representations with expressions of natural language, and (ii) using logical representations of natural language to automate the process of drawing inferences. I will be focussing on (i), and the related issue of defining satisfaction in a model for the semantic representations. By contrast, the important topic of (ii) will not be covered—as yet, there are no theorem provers in NLTK. That said, as pointed out by B&B, for many inference problems in NLP it is desirable to call external and highly sophisticated first-order theorem provers.

One notable feature of B&B is the use of Prolog as the language of implementation. It is not hard to defend the use of Prolog in defining logical representations, given the presence of first-order clauses in Prolog and the fundamental role of resolution in Prolog’s model of computation. Nevertheless, in some circumstances it may be helpful to offer students access to an alternative framework, such as the Python implementation presented here. I also hope that the existence of work in both programming paradigms will turn out to be mutually beneficial, and will lead to a broader community of upcoming researchers becoming involved in the area of computational semantics.
2 Building Semantic Representations

The initial question that we faced in NLTK was how to induce semantic representations for English sentences. Earlier efforts by Edward Loper and Rob Speer had led to the construction of a chart parser for (untyped) feature-based grammars, and we therefore decided to introduce a sem feature to hold the semantics in a parse tree node. However, rather than representing the value of sem as a feature structure, we opted for a more traditional (and more succinct) logical formalism. Since the λ calculus was the pedagogically obvious choice of ‘glue’ language for combining the semantic representations of subconstituents in a sentence, we opted to build on church.py,\(^2\) an independent implementation of the untyped λ calculus due to Erik Max Francis. The NLTK module semantics.logic extends church.py to bring it closer to first-order logic, though the resulting language is still untyped. (1) illustrates a representative formula, translating A dog barks. From a Python point of view, (1) is just a string, and has to be parsed into an instance of the Expression class from semantics.logic.

(1) some x. (and (dog x) (bark x))

The string (dog x) is analyzed as a function application. A statement such as Suzie chases Fido, involving a binary relation chase, will be translated as another function application: ((chase fido) suzie), or equivalently (chase fido suzie). So in this case, chase is taken to denote a function which, when applied to an argument yields the second function denoted by (chase fido). Boolean connectives are also parsed as functors, as indicated by and in (1). However, infix notation for Boolean connectives is accepted as input and can also be displayed.

For comparison, the Prolog counterpart of (1) on B&B’s approach is shown in (2).

(2) some (X, (and (dog X), bark X))

(2) is a Prolog term and does not require any additional parsing machinery; first-order variables are treated as Prolog variables.

(3) illustrates a λ term from semantics.logic that represents the determiner a.

(3) \(\lambda x.\) some x. (and (Q x) (P x))

\(\lambda x\) is the ascii rendering of \(\lambda x.\), and \(\lambda x\) P is shorthand for \(\lambda x.\) P.

For comparison, (4) illustrates the Prolog counterpart of (3) in B&B.

(4) \(\lambda Q\lambda P\). some x. (and (Q x) (P x))

Note that app is used in B&B to signal the application of a λ term to an argument. The right-branching structure for λ terms shown in the Prolog rendering can become fairly unreadable when there are multiple bindings. Given that readability is a design goal in NLTK, the additional overhead of invoking a specialized parser for logical representations is arguable a cost worth paying.

Figure 1 presents a minimal grammar exhibiting the most important aspects of the grammar formalism extended with the sem feature. Since the values of the sem feature have to handed off to a separate processor, we have adopted the convention of enclosing the values in angle brackets, except in the case of variables (e.g., ?subj and ?vp), which undergo unification in the usual way. The app relation corresponds to function application;

In Figure 2, we show a trace produced by the NLTK module parse.featurechart. This illustrates how variable values of the sem feature are instantiated when completed edges are added to the chart. At present, β reduction is not carried out as the sem values are constructed, but has to be invoked after the parse has completed.

The following example of a session with the Python interactive interpreter illustrates how a grammar and a sentence are processed by a parser to produce an object tree; the semantics is extracted from the root node of the latter and bound to the variable e, which can then be displayed in various ways.

```python
>>> gram = GrammarFile.read_file('sem1.cfg')
>>> s = 'a dog barks'
>>> tokens = list(tokenize.whitespace(s))
>>> parser = gram.earley_parser()
>>> tree = parser.parse(tokens)
>>> e = root_semrep(tree)
>>> print e
\(\lambda P.\) some x. (and (Q x) (P x))
>>> print e.simplify()
\(\lambda x.\) some x. (and (Q x) (P x))
>>> print e.simplify().infixify()
some x. (and (dog x) (bark x))
```

Apart from the pragmatic reasons for choosing a functional language as our starting point,
there are also theoretical attractions. It helps introduce students to the tradition of Montague Grammar (Montague, 1974; Dowty et al., 1981), which in turn provides an elegant correspondence between binary syntax trees and semantic composition rules, in the style celebrated by categorial grammar. In the next part of the paper, I will turn to the issue of how to represent models for the logical representations.

3 Representing Models in Python

Although our logical language is untyped, we will interpret it as though it were typed. In particular, expressions which are intended to translate unary predicates will be interpreted as functions of type $e \to \{0, 1\}$ (from individuals to truth values) and expressions corresponding to binary predicates will be interpreted as though they were of type $e \to (e \to \{0, 1\})$. We will start out by looking at data structures which can be used to provide denotations for such expressions.

3.1 Dictionaries and Boolean Types

The standard mapping type in Python is the dictionary, which associates keys with arbitrary values. Dictionaries are the obvious choice for representing various kinds of functions, and can be specialized by user-defined classes. This means that it is possible to benefit from the standard Python operations on dictionaries, while adding additional features and constraints, or in some cases overriding the standard operations. Since we are assuming that our logical language is based on function application, we can readily construct the interpretation of $n$-ary relations in terms of dictionaries-as-functions.

Characteristic functions (i.e., functions that correspond to sets) are dictionaries with Boolean values:

```python
cf = {'d1': True,
      'd2': True,
      'd3': False}
```

cf corresponds to the set $\{d_1, d_2\}$. Since functions are being implemented as dictionaries, func-
tion application is implemented as indexing (e.g., \texttt{cf[‘d1’]} applies \texttt{cf} to argument ‘d1’). Note that \texttt{True} and \texttt{False} are instances of the Python built-in \texttt{bool} type, and can be used in any Boolean context. Since Python also includes \texttt{and} and \texttt{not}, we can make statements (here, using the Python interactive interpreter) such as the following:

\begin{verbatim}
>>> cf[‘d1’] and not cf[‘d3’]
True
\end{verbatim}

As mentioned earlier, relations of higher arity are also modeled as functions. For example, a binary relation will be a function from entities to a characteristic function; we can call these ‘curried characteristic functions’.

\begin{verbatim}
cf2 = {‘d2’: {‘d1’: True},
       ‘d3’: {‘d2’: True}}
\end{verbatim}

cf2 corresponds to the relation \{\{(d1,d2), (d2,d3)\}\}, on two assumptions. First, we are allowed to omit values terminating in \texttt{False}, since arguments that are missing the function will be taken to yield \texttt{False}. Second, as in Montague Grammar, the ‘object’ argument of a binary relation is consumed before the ‘subject’ argument. Thus we write ((\texttt{love} \texttt{m} \texttt{j})) in place of \texttt{love(j,m)}. Recall that we also allow the abbreviated form (\texttt{love} \texttt{m} \texttt{j})

Once we have curried characteristic functions in place, it is straightforward to implement the valuation of non-logical constants as a another dictionary-based class \texttt{Valuation}, where constants are the keys and the values are functions (or entities in the case of individual constants).

While variable assignments could be treated as a list of variable-value pairs, as in B&B, an alternative is again to use a dictionary-based class. This approach makes it relatively easy to impose further restrictions on assignments, such as only assigning values to strings of the form \texttt{x}, \texttt{y}, \texttt{z}, \texttt{x0}, \texttt{x1}, ….

\subsection{3.2 Sets}

Python provides support for sets, including standard operations such as intersection and subset relationships. Sets are useful in a wide variety of contexts. For example, instances of the class \texttt{Valuation} can be given a property \texttt{domain}, consisting of the set of entities that act as keys in curried characteristic functions; then a condition on objects in the \texttt{Model} class is that the domain of some model \texttt{m} is a superset of \texttt{m}’s \texttt{valuation.domain}:

\begin{verbatim}
m.domain.issuperset (m.valuation.domain)
\end{verbatim}

For convenience, \texttt{Valuation} objects have a \texttt{read} method which allows \texttt{n}-ary predicates to be specified as relations (i.e., sets of tuples) rather than functions. In the following example, \texttt{rel} is a set consisting of the pairs (‘d1’, ‘d2’) and (‘d2’, ‘d3’).

\begin{verbatim}
val = Valuation()
rel = set([('d1', 'd2'), ('d2', 'd3')])
val.read([('love', rel)])
\end{verbatim}

\texttt{read} converts \texttt{rel} internally to the curried characteristic function \texttt{cf2} defined earlier.

\section{4 Key Concepts}

\subsection{4.1 Satisfaction}

The definition of satisfaction presupposes that we have defined a first-order language, and that we have a way of parsing that language so that satisfaction can be stated recursively. In the interests of modularity, it seems desirable to make the relationship between language and interpretation less tightly coupled than it is on the approach of B&B; for example, we would like to be able apply similar evaluation techniques to different logical representations. In the current \texttt{NLTK} implementation, the \texttt{nltk_lite.semantics.evaluate} module imports a second module \texttt{logic}, and calls a \texttt{parse} method from this module to determine whether a given Python string can be analysed as first-order formula. However, \texttt{evaluate} tries to make relatively weak assumptions about the resulting parse structure. Specifically, given a parsed expression, it tries to match the structure with one of the following three kinds of pattern:

\begin{verbatim}
(binder, body)
(op, arg_list)
(fun, arg)
\end{verbatim}
Any string which cannot be decomposed is taken to be a primitive (that is, a non-logical constant or individual variable).

A binder can be a λ or a quantifier (existential or universal); an op can be a Boolean connective or the equality symbol. Any other paired expression is assumed to be a function application. In principle, it should be possible to interface the evaluate module with any parser for first-order formulas which can deliver these structures. Although the model checker expects predicate-argument structure as function applications, it would be straightforward to accept atomic clauses that have been parsed into a predicate and a list of arguments.

Following the functional style of interpretation, Boolean connectives in evaluate are interpreted as truth functions; for example, the connective and can be interpreted as the function AND:

\[
\text{AND} = \{\text{True}: \{\text{True}: \text{True}, \text{False}: \text{False}\}, \text{False}: \{\text{True}: \text{False}, \text{False}: \text{False}\}\}
\]

We define OPS as a mapping between the Boolean connectives and their associated truth functions. Then the simplified clause for the satisfaction of Boolean formulas looks as follows:

```python
def satisfy(expr, g):
    if parsed(expr) == (op, args):
        if args == (phi, psi):
            val1 = satisfy(phi, g)
            val2 = satisfy(psi, g)
            return OPS[op][val1][val2]
```

We use 'g[u/x]' to mean that assignment which is just like g except that g(x) = u. In Python, we can build the set sat(φ[x], g) with a for loop:

```python
def satisfies(expr, var, g):
    candidates = []
    if freevar(var, expr):
        for u in domain:
            g.add(u, var)
            if satisfy(expr, g):
                candidates.append(u)
    return set(candidates)
```

An existentially quantified formula ∃x.φ[x] is held to be true if and only if sat(φ[x], g) is nonempty. In Python, len can be used to return the cardinality of a set.

```
elif parsed(expr) == (fun, arg):
    funval = satisfy(fun, g)
    argval = satisfy(psi, g)
    return app(funval, argval)
```

### 4.2 Quantifiers

Examples of quantified formulas accepted by the evaluate module are pretty unexceptional. Some boy loves every girl is rendered as:

'\(\text{some } x.\ ((\text{boy } x) \land \text{all } y.((\text{girl } y) \implies (\text{love } y \ x)))\)'

The first step in interpreting quantified formulas is to define the satisfiers of a formula that is open in some variable. Formally, given an open formula \(\phi[x]\) dependent on \(x\) and a model with domain \(D\), we define the set \(\text{sat}(\phi[x], g)\) of satisfiers of \(\phi[x]\) to be:

\[\{u \in D : \text{satisfy}(\phi[x], g[u/x]) = \text{True}\}\]

We use 'g[u/x]' to mean that assignment which is just like g except that g(x) = u. In Python, we can build the set sat(φ[x], g) with a for loop.

```python
def satisfiers(expr, var, g):
    candidates = []
    if freevar(var, expr):
        for u in domain:
            g.add(u, var)
            if satisfy(expr, g):
                candidates.append(u)
    return set(candidates)
```

An existentially quantified formula \(\exists x.\phi[x]\) is held to be true if and only if sat(φ[x], g) is nonempty. In Python, len can be used to return the cardinality of a set.

```
def satisfiers(expr, var, g):
    candidates = []
    if freevar(var, expr):
        for u in domain:
            g.add(u, var)
            if satisfy(expr, g):
                candidates.append(u)
    return set(candidates)
```
In other words, a formula $\exists x.\phi[x]$ has the same value in model $M$ as the statement that the number of satisfiers in $M$ of $\phi[x]$ is greater than 0.

For comparison, Figure 3 shows the two Prolog clauses (one for truth and one for falsity) used to evaluate existentially quantified formulas in the B&B code modelChecker2.pl. One reason why these clauses look more complex than their Python counterparts is that they include code for building the list of satisfiers by recursion. However, in Python we gain bivalency from the use of Boolean types as return values, and do not need to explicitly mark the polarity of the satisfaction clause. In addition, processing of sets and lists is supplied by a built-in Python library which avoids the use of predicates such as memberList and the [Head|Tail] notation.

A universally quantified formula $\forall x.\phi[x]$ is held to be true if and only if every $u$ in the model’s domain $D$ belongs to $\text{sat}(\phi[x], g)$. The satisfy clause above for existentials can therefore be extended with the clause:

```python
elif parsed(expr) == (binder, body):
  if binder == ('some', var):
    sat = satisfiers(body, var, g)
    return len(sat) > 0
```

In other words, a formula $\forall x.\phi[x]$ has the same value in model $M$ as the statement that the domain of $M$ is a subset of the set of satisfiers in $M$ of $\phi[x]$.

### 4.3 Partiality

As pointed out by B&B, there are at least two cases where we might want the model checker to yield an ‘Undefined’ value. The first is when we try to assign a semantic value to an unknown vocabulary item (i.e., to an unknown non-logical constant). The second arises through the use of partial variable assignments, when we try to evaluate $g(x)$ for some variable $x$ that is outside $g$’s domain. We adopt the assumption that if any subpart of a complex expression is undefined, then the whole expression is undefined. This means that an ‘undefined’ value needs to propagate through all the recursive clauses of the satisfy function. This is potentially quite tedious to implement, since it means that instead of the clauses being able to expect return values to be Boolean, we also need to allow some alternative return type, such as a string. Fortunately, Python offers a nice solution through its exception handling mechanism.

It is possible to create a new class of exceptions, derived from Python’s `Exception` class. The evaluate module defines the class `Undefined`, and any function called by `satisfy` which attempts to interpret unknown vocabulary or assign a value to an out-of-domain variable will raise an `Undefined` exception. A recursive call within `satisfy` will automatically raise an `Undefined` exception to the calling function, and this means that an ‘undefined’ value is automatically propagated up the stack without any additional machinery. At the top level, we wrap `satisfy` with a function `evaluate` which handles the exception by returning the string ‘Undefined’ as value, rather than allowing the exception to raise any higher.

EAFP stands for ‘Easier to ask for forgiveness than permission’. According to van Rossum (2006), “this common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false.” It contrasts with LBYL (‘Look before you leap’), which explicitly tests for pre-conditions (such as type checks) before making calls or lookups. To continue with the discussion of partiality, we can see an example of EAFP in the definition of the `i` function, which handles the interpretation of non-logical constants and individual variables.

```python
try:
  return self.valuation[expr]
except Undefined:
  return g[expr]
```

We first try to evaluate `expr` as a non-logical constant; if `valuation` throws an `Undefined` exception, we check whether `g` can assign a value. If the latter also throws an `Undefined` excep-

---

5This is not the only approach, since one could adopt the position that a tautology such as $p \lor \neg p$ should be true even if $p$ is undefined.
satisfy(Formula,model(D,F),G,pos):-
    nonvar(Formula),
    Formula = some(X,SubFormula),
    var(X),
    memberList(V,D),
    satisfy(SubFormula,model(D,F),[g(X,V)|G],pos).

satisfy(Formula,model(D,F),G,neg):-
    nonvar(Formula),
    Formula = some(X,SubFormula),
    var(X),
    setof(V,memberList(V,D),All),
    setof(V,
        (memberList(V,D),
        satisfy(SubFormula,model(D,F),[g(X,V)|G],neg)),
        All).

Figure 3: Prolog Clauses for Existential Quantification

tion, this will automatically be raised to the calling function.

To sum up, an attractive consequence of this approach in Python is that no additional stipulations need to be added to the recursive clauses for interpreting Boolean connectives. By contrast, in the B&B modelChecker2.pl code, the clauses for existential quantification shown in Figure 3 need to be supplemented with a separate clause for the ‘undefined’ case. In addition, as remarked earlier, each Boolean connective receives an additional clause when undefined.

5 Specifying Models

Models are specified by instantiating the Model class. At initialization, two parameters are called, determining the model’s domain and valuation function. In Table 4, we start by creating a Valuation object val (line 1), we then specify the valuation as a list of constant-value pairs (lines 2–9), using relational notation. For example, the value for ‘adam’ is the individual ‘d1’ (i.e., a Python string); the value for ‘girl’ is the set consisting of individuals ‘g1’ and ‘g2’; and the value for ‘love’ is a set of pairs, as described above. We use the parse method to update val with this information (line 10). As mentioned earlier, a Valuation object has a domain property (line 11), and in this case dom will evaluate to the set set(’d1’, ’b2’, ’g1’, ’g2’, ’d1’). It is convenient to use this set as the value for the model’s domain when it is initialized (line 12). We also declare an Assignment object (line 13), specifying that its domain is the same as the model’s domain.

Given model m and assignment g, we can evaluate m.satisfiers(formula, g), for various values of formulas. This is quite a handy way of getting a feel for how connectives and quantifiers interact. A range of cases is illustrated in Table 5. As pointed out earlier, all formulas are represented as Python strings, and therefore need to be parsed before being evaluated.

6 Conclusion

In this paper, I have tried to show how various aspects of Python lend themselves well to the task of interpreting first-order formulas, following closely in the footsteps of Blackburn and Bos. I argue that at least in some cases, the Python implementation compares quite favourably to a Prolog-based approach. It will be observed that I have not considered efficiency issues. Although these cannot be ignored (and are certainly worth exploring), they are not a priority at this stage of development. As discussed at the outset, our main goal is develop a framework that can be used to communicate key ideas of formal semantics to students, rather than to build systems which can scale easily to tackle large problems.

Clearly, there are many design choices to be made in any implementation, and an alternative framework which overlaps in part with what I have presented can be found in the Python code supplement to (Russell and Norvig, 2003). One important distinction is that the approach adopted here

*http://aima.cs.berkeley.edu/python
val = Valuation()
v = [('adam', 'b1'), ('betty', 'g1'), ('fido', 'd1'), ('girl', set(['g1', 'g2'])), ('boy', set(['b1', 'b2'])), ('dog', set(['d1'])), ('love', set([('b1', 'g1'), ('b2', 'g2'), ('g1', 'b1'), ('g2', 'b1')])),
val.parse(v)
dom = val.domain
m = Model(dom, val)
g = Assignment(dom, {'x': 'b1', 'y': 'g2'})

Figure 4: First-order model \( m \)

<table>
<thead>
<tr>
<th>Formula open in ( x )</th>
<th>Satisfiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(boy x)'</td>
<td>set(['b1', 'b2'])</td>
</tr>
<tr>
<td>'(x = x)'</td>
<td>set(['b1', 'b2', 'g2', 'g1', 'd1'])</td>
</tr>
<tr>
<td>'(boy x) or (girl x)'</td>
<td>set(['b2', 'g2', 'g1', 'b1'])</td>
</tr>
<tr>
<td>'(boy x) and (girl x)'</td>
<td>set([])</td>
</tr>
<tr>
<td>'(love x adam)'</td>
<td>set(['g1'])</td>
</tr>
<tr>
<td>'(love adam x)'</td>
<td>set(['g2', 'g1'])</td>
</tr>
<tr>
<td>'(not (x = adam))'</td>
<td>set(['b2', 'g2', 'g1', 'd1'])</td>
</tr>
<tr>
<td>'some y.(love x y)'</td>
<td>set(['b2', 'g1', 'b1'])</td>
</tr>
<tr>
<td>'all y.(girl y implies (love y x))'</td>
<td>set([])</td>
</tr>
<tr>
<td>'all y.(girl y implies (love x y))'</td>
<td>set(['b1'])</td>
</tr>
<tr>
<td>'(girl x) implies (dog x)'</td>
<td>set(['b1', 'b2', 'd1'])</td>
</tr>
<tr>
<td>'all y.(dog y implies (x = y))'</td>
<td>set(['d1'])</td>
</tr>
<tr>
<td>'(not some y.(love x y))'</td>
<td>set(['b2', 'd1'])</td>
</tr>
<tr>
<td>'some y.(love y adam) and (love x y)'</td>
<td>set(['b1'])</td>
</tr>
</tbody>
</table>

Figure 5: Satisfiers in model \( m \)

is explicitly targeted at students learning computational linguistics, rather than being intended for a more general artificial intelligence audience.

While I have restricted attention to rather basic topics in semantic interpretation, there is no obstacle to addressing more sophisticated topics in computational semantics. For example, I have not tried to address the crucial issue of quantifier scope ambiguity. However, work by Peter Wang (author of the NLTK module nltk_lite.contrib.hole) implements the Hole Semantics of B&B. This module contains a ‘plugging’ algorithm which converts underspecified representations into fully-specified first-order logic formulas that can be displayed textually or graphically. In future work, we plan to extend the semantics package in various directions, in particular by adding some basic inferencing mechanisms to NLTK.

Acknowledgements

I am very grateful to Steven Bird, Patrick Blackburn, Alex Lascarides and three anonymous reviewers for helpful feedback and comments.

References


